ENGR-UH 3511

Computer Organization and Architecture

Name: Nishant Aswani

Net ID: nsa325

Assignment Title: Homework 2

ENGR-UH 3511 Homework 2

Nishant Aswani (nsa325@nyu.edu)

September 8, 2019

Question 2.3

We are given:

- value of i in register $s3
- value of j in register $s4
- base address or &A[0] in register $s6
- base address or &B[0] in register $s7

Hence,

```
sub $t0, $s3, $s4     # i-j stored in register $t0
sll $t0, $t0, 2       # (i-j)*4 to obtain offset, since i and j are 32 bit integers (1 word)
add $t1, $t0, $s6     # add offset to the base address of array A
lw  $t2, 0($t1)       # load the value in A[i-j] into register $t2
sw  $t2, 32($s7)      # save the value in $t2 into B[8] in memory
```

Question 2.4

We assume f and g contain 32-bit integers.

We are given:

- value of f in register $s0
- value of g in register $s1
- base address or &A[0] in register $s6
- base address or &B[0] in register $s7

and the following assembly code, which has been commented to aid in solving the problem:

```
sll  $t0, $s0, 2     # save f*4 into $t0
add  $t0, $s6, $t0   # add $t0 to address of A[0] to get new offset address and save at $t0
sll  $t1, $s1, 2     # save g*4 into $t0
add  $t1, $s7, $t1   # add $t1 to address of B[0] to get new offset address and save at $t1
lw   $s0, 0($t0)     # obtain the value A[f] and save into $s0$
addi $t2, $t0, 4     # save address of A[f+1] into $t2
lw   $t0, 0($t2)     # load value of A[f+1] into $t0
add  $t0, $t0, $s0   # save (value of A[f]) + (value of A[f+1]) into $t0
sw,  $t0, 0($t1)     # save $t0 at B[g]
```

The corresponding C/C++ statement: B[g] = A[f] + A[f+1]

Question 2.18.1

If MIPS were to be implemented with 128 register files, instructions would still need to be implemented with 32 bits, because 32 bits would still remain the word size in a 32 bit architecture. Somehow, all 128 registers would have to be addressable within the 32-bit instruction set.

A traditional register instruction is broken down as follows:

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	5 bits

5 bits are able to address 32 registers, as $2^5 = 32$. To address 128 bits, the instruction set would need to assign 7 bits to rs, rt, and rd, as $2^7 = 128$.

opcode	rs	rt	rd	shamt	funct
? bits	7 bits	7 bits	7 bits	? bits	? bits

In the current implementation, the opcode size is 6 bits, allowing $2^6 = 64$ unique instructions. If the instruction set were to expand to 4 times the current size of 50, according to a Berkeley reference sheet[1], then there would be around 200 instructions. However, the 28[1] R-type instructions all use an opcode of 0, and are then specialized using the funct code.

Assuming expanding the set of instructions would not increase the R-type instructions, we can say that the opcode would have to account for $200 - 28 = 172$ unique instructions. This would require 8 bits for the opcode, which could actually account for potentially 256 instructions.

Thus, in this new implementation, we could remove the R-type instructions' reliance on the funct code and rely on the opcode to distinguish between R-type codes.

opcode	rs	rt	rd	shamt	funct
8 bits	7 bits	7 bits	7 bits	? bits	? bits

Since the func code is rendered useless, the remaining 3 bits could all be assigned to `shamt`.

opcode	rs	rt	rd	shamt	funct
8 bits	7 bits	7 bits	7 bits	3 bits	0 bits

Question 2.18.2

A traditional I-type instruction is broken down as follows:

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

As explained in response to Question 2.18.1, there must be 7 bits to address 128 register files.

Moreover, the opcode convention would have to be retained for all instruction types, hence assigning 8 bits to the opcode.

The remaining 10 bits would be assigned to the immediate code, resulting in the following implementation:

opcode	rs	rt	immediate
8 bits	7 bits	7 bits	10 bits

Question 2.18.3

Implementing the changes from the two responses above, the shift amount would be reduced from 5 bits to 3 bits. So, any shifting requiring more than 3 bits may have to be carried out in smaller chunks, which would increase the size of a MIPS assembly program. However, implementing shift in the format of an I-type instruction may decrease the amount of shift instructions, because it would able to accomodate for larger shifts. This would now be possible because R-type instructions would no longer be differentiated using func codes.

The functionality for immediate instructions would also be stunted, and may have to be carried out in chunks. For example, adding 1025, using `addi`, to a register would require 2 instructions because `addi` would only be able to accommodate 10 bits, and thus 1024 as a maximum value.

On the other hand, for larger programs, having access to more registers could reduce the communication overhead between memory and the CPU and the number of load/save instructions. More registers means fewer data have to be saved to memory to make room for new data.

References

[1] Kevin Liston. *MIPS Reference Sheet*. URL: `https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_help.html`.